DOI:10.1145/2736348

Leslie Lamport

# Viewpoint Who Builds a House without Drawing Blueprints?

*Finding a better solution by thinking about the problem and its solution, rather than just thinking about the code.*

I BEGAN WRITING programs in 1957. For the past four decades I have been a computer science researcher, doing only a small amount of programming. I am the creator of the TLA+ specification language. What I have to say is based on my experience programming and helping engineers write specifications. None of it is new; but sensible old ideas need to be repeated or silly new ones will get all the attention. I do not write safety-critical programs, and I expect that those who do will learn little from this.

Architects draw detailed plans before a brick is laid or a nail is hammered. But few programmers write even a rough sketch of what their programs will do before they start coding. We can learn from architects.

A blueprint for a program is called a *specification*. An architect's blueprint is a useful metaphor for a software specification. For example, it reveals the fallacy in the argument that specifications are useless because you cannot generate code from them. Architects find blueprints to be useful even though buildings cannot be automatically generated from them. However, metaphors can be misleading, and I do not claim that we should write specifications just because architects draw blueprints.

IMAGE BY SAKONBOON SANSRI

The need for specifications follows from two observations. The first is that it is a good idea to think about what we are going to do before doing it, and as the cartoonist Guindon wrote: "Writing is nature's way of letting you know how sloppy your thinking is."

We think in order to understand what we are doing. If we understand something, we can explain it clearly in writing. If we have not explained it in writing, then we do not know if we really understand it.

The second observation is that to write a good program, we need to think above the code level. Programmers spend a lot of time thinking about how to code, and many coding methods have been proposed: test-driven development, agile programming, and so on. But if the only sorting algorithm a programmer knows is bubble sort, no such method will produce code that sorts in $O(n \log n)$ time. Nor will it turn an overly complex conception of how a program should work into simple, easy to maintain code. We need to understand our programming task at a higher level before we start writing code.

Specification is often taken to mean something written in a formal language with a precise syntax and (hopefully) a precise semantics. But formal specification is just one end of a spectrum. An architect would not draw the same kind of blueprint for a toolshed as for a bridge. I would estimate that 95% of the code programmers write is trivial enough to be adequately specified by a couple of prose sentences. On the other hand, a distributed system can be as complex as a bridge. It can require many specifications, some of them formal; a bridge is not built from a single blueprint. Multithreaded and distributed programs are difficult to get right, and formal specification is needed to avoid synchronization errors in them. (See the article by Newcombe et al. on page 66 in this issue.)

The main reason for writing a formal spec is to apply tools to check it. Tools cannot find design errors in informal specifications. Even if you do not need to write formal specs, you should learn how. When you do need to write one, you will not have time to learn how. In the past dozen years, I have written formal specs of my code about a half dozen times. For example, I once had to write code that computed the connected components of a graph. I found a standard algorithm, but it required some small modifications for my use. The changes seemed simple enough, but I decided to specify and check the modified algorithm with TLA+. It took me a full day to get the algorithm right. It was much easier to find and fix the errors in a higher-level language like TLA+ than it would have been by applying ordinary program-debugging tools to the Java implementation. I am not even sure I would have found all the errors with those tools.

**We need to understand our programming task at a higher level before we start writing code.**

Writing formal specs also teaches you to write better informal ones, which helps you think better. The ability to use tools to find design errors is what usually leads engineers to start writing formal specifications. It is only afterward that they realize it helps them to think better, which makes their designs better.

There are two things I specify about programs: what they do and how they do it. Often, the hard part of writing a piece of code is figuring out what it should do. Once we understand that, coding is easy. Sometimes, the task to be performed requires a nontrivial algorithm. We should design the algorithm and ensure it is correct before coding it. A specification of the algorithm describes how the code works.

Not all programs are worth specifying. There are programs written to learn something—perhaps about an interface that does not have an adequate specification—and are then thrown away. We should specify a program only if we care whether it works right.

Writing, like thinking, is difficult; and writing specifications is no exception. A specification is an abstraction. It should describe the important aspects and omit the unimportant ones. Abstraction is an art that is learned only through practice. Even with years of experience, I cannot help an engineer write a spec until I understand her problem. The only general rule I have is that a specification of what a piece of code does should describe everything one needs to know to use the code. It should never be necessary to read the code to find out what it does.

There is also no general rule for what constitutes a "piece of code" that requires a specification. For the pro-

## Calendar of Events

**April 13–16**
**CPS Week '15: Cyber Physical Systems Week 2015,**
**Co-Sponsored: Other Societies,**
**Seattle, WA,**
**Contact: Jie Liu,**
**Contact Email: liuj@microsoft.com**

**April 13–17**
**Symposium on Applied Computing, Salamanca, Spain,**
**Sponsored: ACM/SIG,**
**Contact: Roger Wainwright,**
**Contact Email: rogerw@utulsa.edu**

**April 14–16**
**HSCC'15: 18th International Conference on Hybrid Systems: Computation and Control (part of CPS Week),**
**Seattle, WA,**
**Sponsored: ACM/SIG,**
**Contact: Sriram Sankaranarayanan,**
**Contact Email: srirams@gmail.com**

**April 14–16**
**ICCPS '15: ACM/IEEE 6th International Conference on Cyber-Physical Systems (with CPS Week 2015),**
**Seattle, WA,**
**Co-Sponsored: Other Societies,**
**Contact: Ian Mitchell,**
**Contact Email: mitchell@cs.ubc.ca**

**April 14–16**
**IPSN '15: The 14th International Conference on Information Processing in Sensor Networks (co-located with CPS Week 2015),**
**Seattle, WA,**
**Co-Sponsored: Other Societies,**
**Contact: Bhaskar Krishnamachari,**
**Contact Email: bkrishna@usc.edu**

**April 18–23**
**CHI '15: CHI Conference on Human Factors in Computing Systems,**
**Seoul, Republic of Korea,**
**Sponsored: ACM/SIG,**
**Contact: Jinwoo Kim,**
**Contact Email: create2gether@gmail.com**

**April 21–24**
**EuroSys '15: Tenth EuroSys Conference 2015,**
**Bordeaux, France,**
**Sponsored: ACM/SIG,**
**Contact: Laurent Reveillere,**
**Contact Email: reveillere@labri.fr**

gramming I do, it may be a collection of fields and methods in a Java class, or a tricky section of code within a method. For an engineer designing a distributed system, a single spec may describe a protocol that is implemented by code in multiple programs executed on separate computers.

Specification should be taught in school. Some universities offer courses on specification, but I believe that most of them are about formal specification languages. Anything they teach about the art of writing real specs is an accidental by-product. Teachers of specification should write specifications of their own code, as should teachers of programming.

Computer scientists believe in the magical properties of language, and a discussion of specification soon turns to the topic of specification languages. There is a standard language, developed over a couple of millennia, for describing things precisely: mathematics. The best language for writing informal specifications is the language of ordinary math, which consists of precise prose combined with mathematical notation. (Sometimes additional notation from programming languages can be useful in specifying how a program works.) The math needed for most specifications is quite simple: predicate logic and elementary set theory. This math should be as natural to a programmer as numbers are to an accountant. Unfortunately, the U.S. educational system has succeeded in making even this simple math frightening to most programmers.

Math was not developed to be checked by tools, and most mathematicians have little understanding of how to express things formally. Designers of specification languages usually turn to programming languages for inspiration. But architects do not make their blueprints out of bricks and boards, and specifications should not be written in program code. Most of what we have learned about programming languages does not apply to writing specifications. For example, information hiding is important in a programming language. But a specification should not contain lower-level details that need to be hidden; if it does, there is something wrong with the language in which it is written. I believe the closer a specification language comes to ordinary mathematics, the more it aids our thinking. A language may have to give up some of the elegance and power of math to provide effective tools for checking specs, but we should have no illusion that it is improving on ordinary mathematics.

Programmers who advocate writing tests before writing code often believe those tests can serve as a specification. Writing tests does force us to think, and anything that gets us to think before coding is helpful. However, writing tests in code does not get us thinking above the code level. We can write a specification as a list of high-level descriptions of tests the program should pass—essentially a list of properties the program should satisfy. But that is usually not a good way to write a specification, because it is very difficult to deduce from it what the program should or should not do in every situation.

Testing a program can be an effective way to catch coding errors. It is not a good way to find design errors or errors in the algorithm implemented by the program. Such errors are best caught by thinking at a higher level of abstraction. Catching them by testing is a matter of luck. Tests are unlikely to catch errors that occur only occasionally—which is typical of design errors in concurrent systems. Such errors can be caught only by proof, which is usually too difficult, or by exhaustive testing. Exhaustive testing—for example, by model checking—is usually possible only for small instances of an abstract specification of a system. However, it is surprisingly effective at catching errors—even with small models.

The blueprint metaphor can lead

**If we do not start with a specification, every line of code we write is a patch.**

**Alignment example.**

| User Input | Naive Formatting | Desired Formatting |
|---|---|---|
| `a   =  b` | $a = b$ | $a \quad = b$ |
| `/\ ccc >= d` | $\land\ ccc \geq d$ | $\land\ ccc \geq d$ |

us astray. Blueprints are pictures, but that does not mean we should specify with pictures. Anything that helps us think is useful, and pictures can help us think. However, drawing pictures can hide sloppy thinking. (An example is the classic plane-geometry "proof" that all triangles are isosceles.) Pictures usually hide complexity rather than handling it by abstraction. They can be good for simple specifications, but they are not good for dealing with complexity. That is why flowcharts were largely abandoned decades ago as a way to describe programs.

Another difference between blueprints and specifications is that blueprints get lost. There is no easy way to ensure a blueprint stays with a building, but a specification can and should be embedded as a comment within the code it is specifying. If a tool requires a formal specification to be in a separate file, a copy of that file should appear as a comment in the code.

In real life, programs often have to be modified after they have been specified—either to add new features, or because of a problem discovered during coding. There is seldom time to rewrite the spec from scratch; instead the specification is updated and the code is patched. It is often argued that this makes specifications useless. That argument is flawed for two reasons. First, modifying undocumented code is a nightmare. The specs I write provide invaluable documentation that helps me modify code I have written. Second, each patch makes the program and its spec a little more complicated and thus more difficult to understand and to maintain. Eventually, there may be no choice but to rewrite the program from scratch. If we do not start with a specification, every line of code we write is a patch. We are then building needless complexity into the program from the beginning. As Dwight D. Eisenhower observed: "No battle was ever won according to plan, but no battle was ever won without one."

Another argument against specification is that the requirements for a program may be too vague or ill-defined to be specified precisely. Ill-defined requirements mean not that we do not have to think, but that we have to think even harder about what a program should do. And thinking means specifying. When writing the pretty-printer for TLA+, I decided that instead of formatting formulas naively, it should align them the way the user intended (see the accompanying figure).

It is impossible to specify precisely what the user intended. My spec consisted of six alignment rules. One of them was:

> If token *t* is a left-comment token, then it is left-comment aligned with its covering token.

where terms like *covering token* are defined precisely but informally. As I observed, this is usually not a good way to write a spec because it is hard to understand the consequences of a set of rules. So, while implementing the rules was easy, debugging them was not. But it was a lot easier to understand and debug six rules than 850 lines of code. (I added debugging statements to the code that reported what rules were being applied.) The resulting program does not always do the right thing; no program can when the right thing is subjective. However, it works much better, and took less time to write, than had I not written the spec. I recently enhanced the program to handle a particular kind of comment. The spec made this a simple task. Without the spec, I probably would have had to recode it from scratch. No matter how ill-defined a problem may be, a program to solve it has to do something. We will find a better solution by thinking about the problem and its solution, rather than just thinking about the code.

A related argument against specification is that the client often does not know what he wants, so we may as well just code as fast as we can so he can tell us what is wrong with the result. The blueprint metaphor easily refutes that argument.

The main goal of programmers seems to be to produce software faster, so I should conclude by saying that writing specs will save you time. But I cannot. When performing any task, it is possible to save time and effort by doing a worse job. And the result of forcing a programmer to write a spec, when she is convinced that specs are a waste of time, is likely to be useless—just like a lot of documentation I have encountered. (Here is the description of the method *resetHighlightRange* in a class *TextEditor*: "Resets the highlighted range of this text editor.")

To write useful specifications, you must want to produce good code—code that is easy to understand, works well, and has few errors. You must be sufficiently motivated to be willing to take the time to think and specify before you start coding. If you make the effort, specification can save time by catching design errors when they are easier to fix, before they are embedded in code. Formal specification can also allow you to make performance optimizations that you would otherwise not dare to try, because tools for checking your spec can give you confidence in their correctness.

There is nothing magical about specification. It will not eliminate all errors. It cannot catch coding errors; you will still have to test and debug to find them. (Language design and debugging tools have made great progress in catching coding errors, but they are not good for catching design errors.) And even a formal specification that has been proved to satisfy its required properties could be wrong if the requirements are incorrect. Thinking does not guarantee that you will not make mistakes. But not thinking guarantees that you will. **C**

**Leslie Lamport** is a principal researcher at Microsoft Research and recipient of the 2013 ACM A.M. Turing Award.